AFRL-IF-RS-TR-2005-227
Final Technical Report
June 2005

AGENT SEMANTIC COMMUNICATION SERVICES

Teknowledge Corp.

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. K527

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2005-227 has been reviewed and is approved for publication

APPROVED: /s/
CRAIG S. ANKEN
Project Engineer

FOR THE DIRECTOR: /s/
JOSEPH CAMERA, Chief
Information & Intelligence Exploitation Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 2005	Final Aug 00 – Dec 04

4. TITLE AND SUBTITLE

AGENT SEMANTIC COMMUNICATION SERVICES

5. FUNDING NUMBERS

C - F30602-00-C-0171
PE - 62301E
PR - DAML
TA - 00
WU - 02

6. AUTHOR(S)

John Li and Allan Terry

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Teknowledge Corporation
1800 Embarcadero Road
Palo Alto, CA 94303

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/IFED
525 Brooks Road
Rome, NY 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-2005-227

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Craig S. Anken, IFED, 315-330-2074, Craig.Anken@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***

Agent Semantic Communication Services (ASCS) addressed a fundamental issue in the adoption and use of semantic technologies. It provided a semantic search engine with Web-scale architecture. Semantic search answers a logical query by return bindings for the unbound variables in the query. In information retrieval approaches such as Google, a large number of URIs are returned that contain one or more of the search keywords, ranked by an estimate of how relevant they are to the user's query. The simplistic ASCS search is based on a single search agent. The agent uses a combination of lookup and inference techniques to return answers from its repository. The full ASCS search is based on a network of distributed agents. The ASCS project created one such search agent. The final version was checked into SemWebCentral.org as the OWL Semantic Search Services project. The OWL Semantic Search Service is a good tool for trying disparate data sources together. It is not restricted to the Web. ASCS is ready for transition to early adopters of semantic technologies.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Semantic Web, Search, Agents, inference	33
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Contents

Figures

Executive Summary

This project addressed a fundamental issue in the adoption and use of semantic technologies. Given data resources marked up with OWL, the agent's access to data remains as a core problem. The agent has a query—what data answers it? In most cases, the answer requires data to be combined from several sources. The data are expressed using many ontologies or namespaces—how does the agent obtain an answer that is meaningful? Most of the data are very dynamic and uncontrolled, especially in the Semantic Web—how does the agent exploit this resource in a timely and scalable way?

Teknowledge's Agent Semantic Communication Services (ASCS) project addressed these issues by providing a semantic search engine with Web-scale architecture. Semantic search answers a logical query by returning bindings for the unbound variables in the query. The user obtains one or more very specific answers to their query. Contrast this to information-retrieval approaches such as Google. These return a large number of URIs to resources that contain one or more of the search keywords, ranked by an estimate of how relevant they are to the user's query. The examination of each URI to see if it does, in fact, contain an answer is an exercise left to the user.

The simplest ASCS search is based on a single search agent. Each such agent indexes the content it has found or has been given. This content is expressed using one or more known ontologies. The agent uses a combination of lookup and inference techniques to return answers from its repository.

The full ASCS search is based on a network of distributed agents. Search agents can often be thought of as specialists in specific topics or specific ontologies. Translation agents mediate between different ontologies. They reformulate a query in terms understandable by another search agent, and translate the results into the terms of the original querying agent. Search uses a form of spreading activation algorithm that locates other appropriate search agents using Web services discovery and matchmaking. The process is guided by techniques for controlling resource use, preventing loops, and combining results.

The ASCS project created one such search agent. The final version was checked into SemWebCentral.org as the OWL Semantic Search Services project. While the main users of semantic search will be intelligent agents, humans are the primary users in the short term. Accordingly, we have provided three different GUIs addressing different types of users.

- http://plucky.teknowledge.com/daml/damlquery.jsp represents a basic interface for the technical user.
- http://oak.teknowledge.com:8080/daml/damlquery.jsp is intended for power users who need more control of the query.
- http://ibis.teknowledge.com:8080/daml/query.jsp is an experiment in natural language query for the non-logician.

The semantic search service has been available to the public 7x24 for over two years. It currently indexes about 8 million OWL triples and answers most queries within 1-10 seconds on a modest server.

The automated translation or mapping between two ontologies is a very difficult research issue; one which is not yet solved. Our strategy was to harvest the best work of the DAML teams working on this problem as it became available. In the meantime, we

adopted two approaches to the translation function. We created a semi-automated ontology mapper based on lexical techniques. Using this mapper, we created correspondences between 16 sample ontologies and our SUMO-MILO ontology. This hub-and-spoke approach was used in the Ibis system to expand the search beyond a single namespace. In the other two systems, we provided a search-broadening capability to reduce the dependence of results on the exact wording of the query. The user may request inference along OWL equivalence, generalization/specialization, or inverse relations.

The ASCS project leaves behind many achievements and working components:

- The OWL Semantic Search Service, a complete search component including crawling, indexing, and query.
- Java and Web services interfaces for access by other programs.
- Three different GUIs that help human users formulate queries.
- A publish/subscribe capability that promotes dynamic update and situation awareness. Content creators can publish their URIs to ASCS for immediate indexing. Users can subscribe queries and specify whether they are to be run on a schedule or by event (such as on data change). When new results are available, they are emailed to the subscriber.
- A semi-automated, extensible, lexically based ontology mapping utility. This mapper came in second in the 2004 I^3COM trials, showing particular strength in the handling of large ontologies.
- We contributed a major ontology resource to help seed the Semantic Web. We posted OWL versions of our SUMO upper ontology (now in the IEEE standardization process), our MILO midlevel ontology, and 13 major domain ontologies aligned with SUMO and MILO. The mappings into SUMO mentioned above are also available.
- An initial implementation of the distributed agent architecture.

The search service code, the ontology mapper, and the ontologies have all been posted to SemWebCentral.org.

The OWL Semantic Search Service is a good tool for tying disparate data sources together. It is not restricted to the Web. Web pages may contain OWL markup, but so can other unstructured data sources like text documents and email. Structured data sources such as databases can export their data marked up in OWL based on a mapping to their schema. An excellent next step for this program will be as a search service / data integrator within one organization. Given that we index 8M triples on one decidedly ordinary server, this should be a good fit. When a network of servers is required, our distributed architecture can be employed to scale up. If a hub-and-spoke model is adopted, ontology mapping can take place offline and the translation agents are not needed. ASCS is ready for transition to early adopters of semantic technologies.

Project objectives

The Semantic Web received significant impetus from the DARPA DAML program. The Teknowledge DAML project addresses a critical infrastructure need of the Semantic Web—access to information. Without timely and scalable access to information in the form of semantic markup, the tools and agents that process the markup have greatly reduced value. The World Wide Web has similar problems, and the familiar search engines are now considered essential to the utility of the Web. The Semantic Web and the larger domain of marked-up data has its particular needs:

- Data with markup can reside anywhere and is not restricted to Web pages or documents that can appear on the Web.
- The presence of markup containing semantic information enables queries that return real answers instead of only pointers to where one may explore for answers.
- Answering questions generally involves combining information from many sources, but there is no guarantee any two sources share the same namespace or ontology. There is no guarantee any two OWL triples share the same namespace, even if both come from the same source.
- Generating answers to questions is harder than simply searching for documents. Data must be logically combined and often some form of inference is required.

Any component for semantic data access has many of the same requirements as a search engine on the traditional Web. Whether it does inference or not, users still expect very fast response. The component must be highly scalable, capable of answering questions from structured data sources such as databases, and from Web-scale collections of documents containing markup. Finally, queries and their answers must be stated in the user's terms, which generally vary from those used in the data.

Technical discussion

The Teknowledge ASCS project provides a semantic search engine to address these needs. The OWL Semantic Search Service (OSSS) focuses on answering questions for intelligent agents or for humans. It is called a search service because of the similarities to traditional search engines: it crawls to discover content, it indexes the content, and it answers queries posed to its repository. It differs in the queries it accepts and how they are processed. Google et al follow an information retrieval approach that seeks pages that contain one or more of the search terms. Many answers are returned but mostly irrelevant as no semantics are employed. Complex relevance ranking algorithms are computed in an attempt to mitigate this problem.

OSSS supports conjunctive logical queries stated in OWL. For example, "What suicide bombings occurred in any sub-region of the Middle East?" translates to:

```
(?EVENT, located, ?PLACE)
(?EVENT, type, SuicideBombing)
(?PLACE, geographicSubregion, MiddleEastRegion)
```

The answer is returned as bindings for the variables in the query (?EVENT and ?PLACE in this example). There may be multiple valid binding sets representing different answers, but all are relevant. There are no false-positive answers.

Architecture

The full ASCS architecture is based on distributed agents, as shown in Figure 1. Each semantic search agent (SSA) indexes some content and can answer queries about it. The content is not limited to Web pages. It can be any data that includes OWL markup. Any SSA knows only what it indexes and may not be able to answer all parts of the query by itself. Any SSA can choose to seek other SSAs that can contribute to the answer. Any SSA that asks others for assistance also has functionality for controlling search and combining results.

Semantic translation service (STS) agents mediate between different ontologies. An STS can translate a query between the ontology or ontologies used by the questioner into the ontology used by the answering SSA, and then translate the answers back into the initial terms and concepts. There is no need for any specific STS to be able to translate any arbitrary ontology into any other; we assume STS agents will be specialists.

Agents find each other via UDDI and interact via Web Services. Service discovery asks "Can you help with this topic?" and also asks what ontologies the SSA supports. When two SSAs use the same ontology, they can query each other freely. STS agents are needed only when the two SSAs use different ontologies, necessitating translation of queries and results between them.

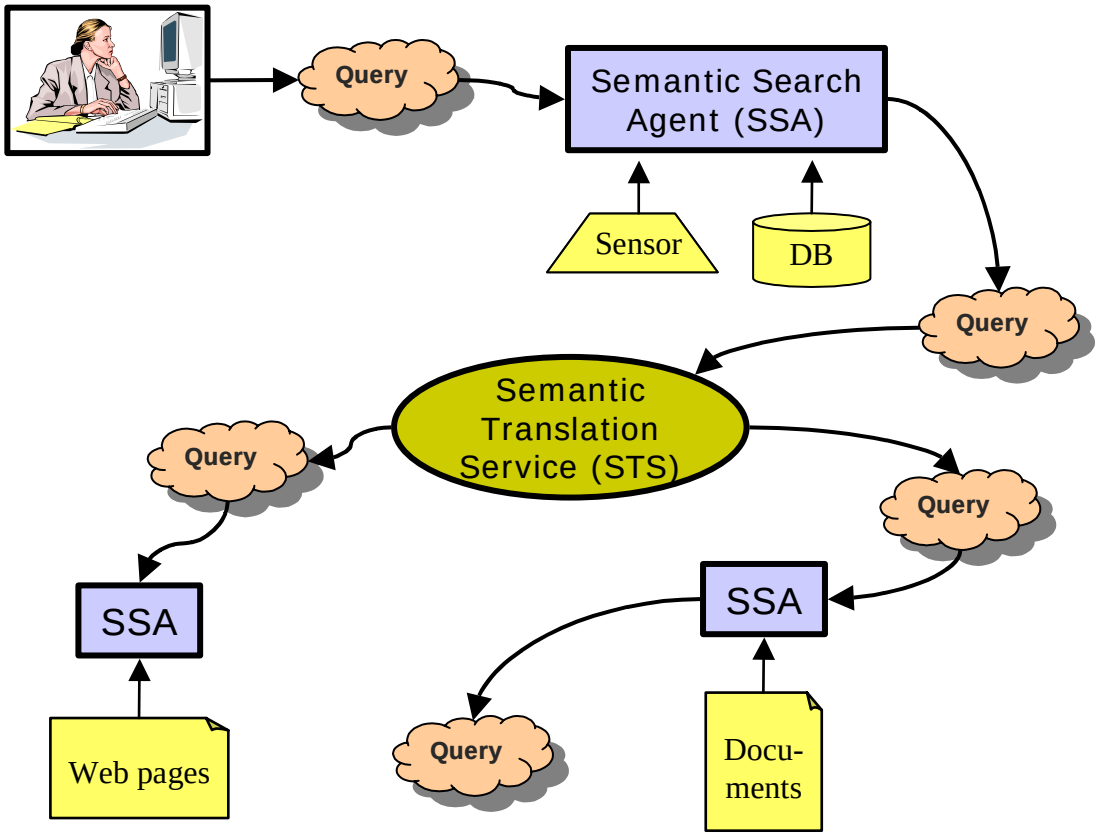

Figure 1. An example of distributed search of heterogenous data sources

Figure 2 shows the interaction in more detail. The original query comes from a user interface or intelligent agent. Either way, the query will be in OWL, expressed in some ontology or ontologies. SSA-1 makes the first attempt to answer the question. Perhaps it can find answers in the triples within its repository, obtained by crawling, or perhaps it decides further work would be fruitful. It may send the whole query on to other SSAs or it may split off just one part of the query. Very occasionally, it will find another SSA, with appropriate content, that uses the same terms and concepts that it does. The query can be passed on as-is in that instance. Most often, it will find a SSA with appropriate content but using different ontologies. An appropriate STS is then needed to translate. SSA-14 might index a repository of sensor readings that use a process-chemistry ontology. STS-B then needs to translate in and out of this ontology into concepts that SSA-1 can understand. STSs can also potentially translate to SSAs that are not OWL based, as long as they are based on some ontology. SSA-8 is basically a database maintained by some enterprise process. Instead of exporting selected tables and columns to OWL for some SSA to index, use of STS-8 and the search API allows it to be treated as an SSA. STS-8 translates the queries from SSA-1 into SQL and translates the results into triples.

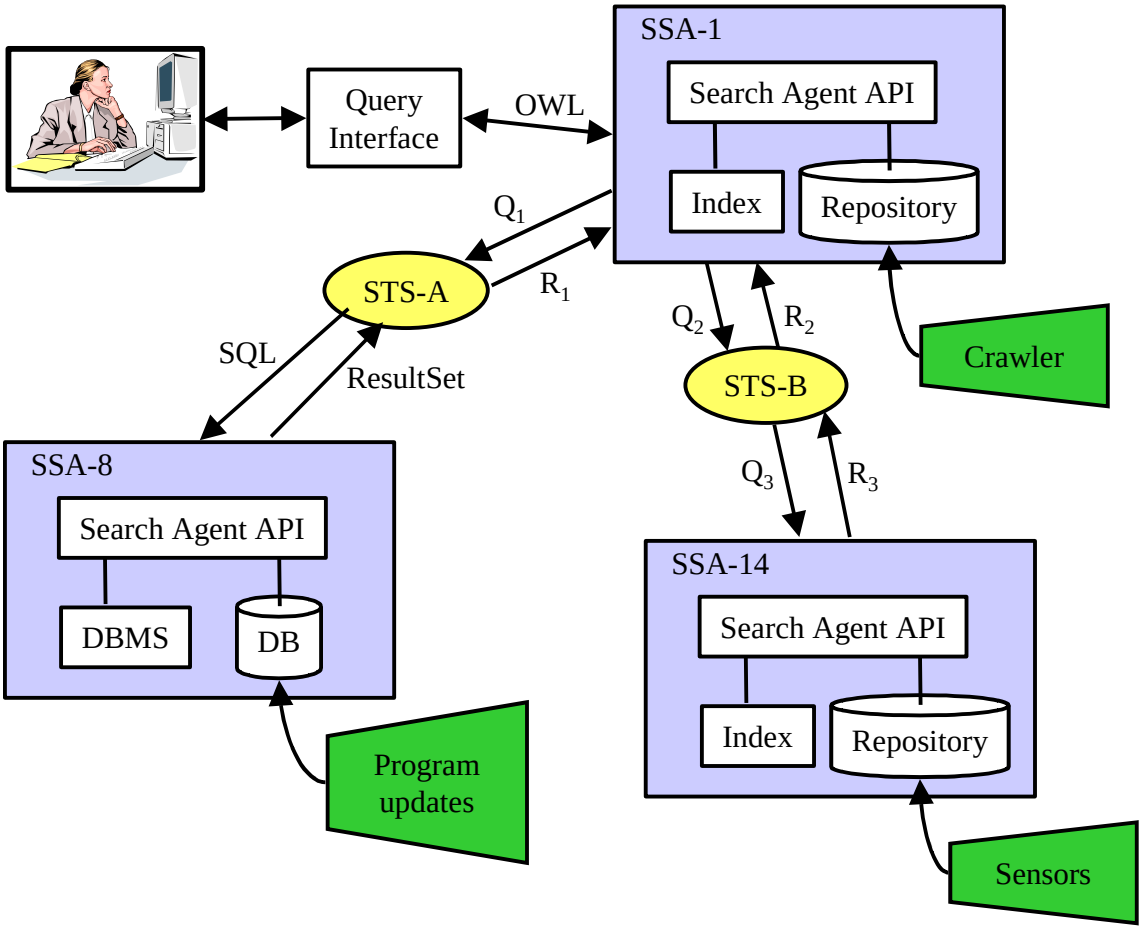

Figure 2. ASCS agent architecture

The implementation of this initial version of ASCS was in Java and XSB prolog. Prolog was selected for the inference we expected to do in both SSAs and STSs. XSB was specifically chosen for its database connectivity and for its extensions to standard Prolog expressiveness (such as tabling and well-founded semantics). Each SSA could search their own repository and query others. Control of distributed search was provided by tracking nodes for loop avoidance, and by imposing a timeout (time to live) on the search. When a SSA needed to query another, it can use the matchmaking services provided by the third party such as CMU's Atlas matchmaker.

This work provided a good early check on the architecture. However, Atlas (and Web services in general) was in an early stage and the DAML program as a whole decided to focus on standards such as DAML-S and OWL-S more than bringing along these components in the short run. We decided to further develop the two agent components and then revisit the full architecture later.

An example ASCS application

One of our test systems shows an example of a small ASCS. TekPortal is Teknowledge's middleware product for the financial industry. It aggregates financial data for customers with online accounts at banks, brokers, and other financial institutions. The data is gathered into one database where it can be displayed and analyzed. For the test system, TekPortal and real-world data sources are combined using ASCS.

The goal was to provide data for a simple financial advisor program. Customer data such as net worth, age, assets and liabilities, and other background factors are obtained from the TekPortal database. We created an exporter from selected tables to DAML (this was before OWL) using our financial domain ontology. We also took two data sources—Bloomberg government bond data for the G7 countries and an Indian Government bond data source—and marked them up with DAML. A small translation agent is needed for each source to convert the data, as presented, into the format and concepts used by our financial ontology. Once all the data is expressed in DAML, the SSA supporting the financial advisor can query it. It can be argued that this is a simple data-integration problem, and that once data sources are brought into a standard DB format (instead of the DAML tuples we used), this is a straight SQL problem. The point of the system was to show that an ontological approach provides a firm basis for the efficient semantic integration, and only requires three mappings in a hub-and-spoke approach. The system also showed that the financial advisor may ask more sophisticated queries than SQL allows, by using the query broadening features discussed below. As the number of sources and ontologies increases, the traditional SQL approach begins to break down. The ASCS approach is more flexible in representation (a repository of DAML tuples instead of a fixed DB schema) and in the type of queries that are supported.

A Semantic Search Agent

After initial experiments to validate the overall architecture, we focused on building a substantial SSA. Our goals for version 2 of the SSA were:

- Build a component for the DAML community that handles real-world scruffiness and complexity.
- Explore issues of how to formulate queries, especially explore GUIs that help users in this.
- Find out how much capacity a single SSA can handle by itself.
- Find solutions to issues of scale, in size of repository and number of users.

The detailed structure of OSSS, the current incarnation of a SSA, is shown in Figure 3. A back-end system handles crawling and indexing. A separate runtime component does the work of answering queries. The two are tied together by a database that holds the repository.

Back-end support

The back-end system has much in common with traditional search engines. We built a simple spider that looks for Web pages with semantic markup in RDF, DAML, DAML+OIL, or OWL. We started from a seed list or URLs from DAML.org, and add to it each time we do a new crawl.

We quickly encountered real-world scruffiness. Sites point to each other multiple times in a page, and there are sets of pages that form tangled loops. There are often multiple forms of the same URL on a page and in many pages, the HTML is ill-formed and difficult for our parser to deal with. Many sites are not available when we visit, necessitating a return visit later and perhaps removal from the list. For these problems and many others, we developed the crawler far enough to collect several million triples, but did not devote many resources in solving crawler issues. The URL Scrubber component handles much of the clean up, removal of duplicates, and checking of collected results. This is a very important step that prevents "garbage in".

A major problem is that we frequently find many different syntaxes designating the same location. For example, a relative address (./resource/file.owl), a full URL (http://www.somewhere.com/public/resource/file.owl) and a namespace declaration within the file (test=url#). Most OWL parsers do not try or are unable to unify these variants. The URL scrubber implements a simple method of unifying namespaces. It identifies the non-physical-filename references by comparing them to the list of previously crawled URLs, and then unifying the namespace with URLs on the list if they are sufficiently "close". Two items are close if the only difference between the references is the presence or absence of file extensions and some other features. This simple heuristic greatly reduces the number of available namespaces and makes the items in the list more unique.

Web pages with markup typically have many triples per page. Some pages are essentially databases with a large number of triples. However, OSSS indexes individual triples, not pages. The indexer uses HP's Jena 2 to parse the markup. Jena produces 4-tuples of (URL, subject, relation, object). The indexer maintains a table of URLs so that a simple index can be used in place of long URL strings. It also maintains the table of triples, stored in the DB as a 7-tuple consisting of the URL-index and a namespace/item

pair for each element of the OWL triple. These namespaces can also be stored as indices to the URL table, for further reduction in space.

We changed the back-end system from batch to incremental operation in the last year. This accommodates the "publish" feature and the desire to further automate the crawling and indexing process. At the same time, we also re-implemented the repository in a relational database—the maxDB version of mySQL. We are now able to dynamically index while the runtime system is in operation, and transactionally update the runtime repository.

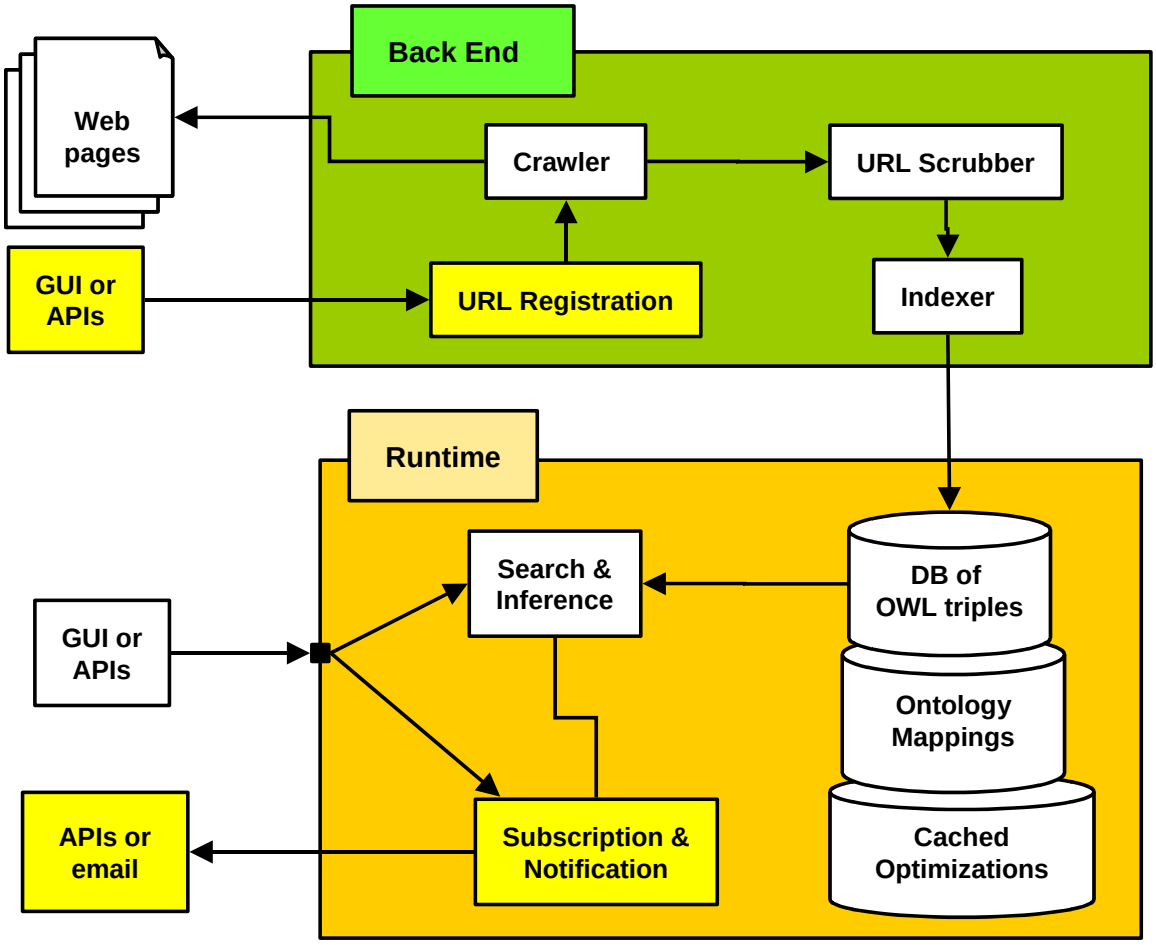

Figure 3. OSSS structure

The indexer accomplishes a second task. As an aid to GUIs and to satisfy a specific user's request, we maintain a string completion facility for terms in namespaces used in the repository. For example, a user may not remember that the relation they want to use is called works-for, work-for, or worksFor. The string completion allows a GUI to show completions of "work" to remind the user which term is correct in a selected ontology. This index is implemented as a prefix tree using the first ten characters. Ten was chosen as a good balance between accuracy and tree size after experiments with different length settings. The tree needs to be in RAM for decent performance because of

the heavy amount of look up. In a repository of 8M triples, the prefix tree for 10 characters is about 400 Mb. For 30 characters, it is about 1 Gb.

The runtime query component

This is the active part of OSSS that answers queries. OSSS accepts queries in the form of conjuncts of OWL triple clauses. At its most simple level, the search engine is a standard Prolog search through a knowledge base of 7-tuples to find bindings for the free variables in the query. Results are returned in batches of N responses. If more are wanted, the GUI or API is able to utilize Prolog's capability to continue searching. This is both a user convenience and a performance aid. For an example of the latter, consider that a small number of terms are grossly over-represented in the OWL markup that presently exists. A wildcard query with "type" as the predicate can return over 500,000 records. The terms "comment", "label", "domain", and "range" are all similarly productive. Limiting responses to groups for 25 or so yields more predictable query latency when the number of answers can vary from one to a million.

Query optimization

The most important development that has significantly enhanced the performance of our server is a new query execution algorithm we designed and developed for the expensive queries. Before we deployed the algorithm, the response time for some "expensive" queries was a showstopper, despite our efforts with traditional query optimization measures. Most of these "expensive" queries consist of multiple clauses. A natural solution would be to rearrange the order of the clauses based on heuristics, such as the numbers of constants and variables in the clauses, the ratio between them, and the frequencies of the given constants, etc. We call this method the *initial static analysis (ISA)* of the query since the order rearrangement of the clauses is done before the query execution and based on the heuristics about the given constants in the original query. Our new algorithm runs the query in a different way. It executes one clause per step. In each step it evaluates the remaining clauses and executes the one best matching the heuristics. We call our new algorithm the *continuous dynamic analysis (CDA)* of the query because the order of the clauses in a query is now optimized continuously based on the best dynamic information the server can find in each step of the execution. The following is a real example we tested in an early version of our ASCS server that explained the different impacts of ISA and CDA on the query response time.

The query is a wildcard search: "Who are the daughters of George_VI /Windsor/?" According to the ontology that defines the data (http://www.daml.org/2001/01/gedcom/gedcom.daml), it is formalized into a simplified conjunctive form of triples containing only a subject, a predicate and an object:
Query 0:

```
(?Person, name, 'George_VI /Windsor/')
(?Person, spouseIn, ?Family)
(?Daughter, childIn, ?Family)
(?Daughter, sex, F)
(?Daughter, name, ?Name).
```

For all constants in the query (the ones without ?), their first-order frequencies of occurrence are given as follows. The notation "freq(word, role) = n" means the "word" as

the “role” has n occurrences in our DAML server (note that we only consider the first-order frequencies).

```
freq(name, predicate) = 117,499
freq(sex, predicate) = 23,000
freq(spouseIn, predicate) = 16,643
freq(childIn, predicate) = 14,721
freq('F', object) = 15,438
freq('George_VI /Windsor/', object) = 1
```

Suppose our heuristics prefers clauses of more constants as well as of constants of lower frequency. Based on the heuristics and the given first-order frequencies, Query 1 is the “optimal” search order for this query since the first two clauses both have only one variable.

Query 1:

```
(?Person, name, 'George_VI /Windsor/'),
(?Daughter, sex, F),
(?Daughter, childIn, ?Family),
(?Person, spouseIn, ?Family),
(?Daughter, name, ?Name).
```

If we change our heuristics to prefer only clauses containing lower frequency constants, we'll get the following sequence:

```
(?Person, name, 'George_VI /Windsor/'),
(?Daughter, childIn, ?Family),
(?Daughter, sex, F),
(?Person, spouseIn, ?Family),
(?Daughter, name, ?Name).
```

However, there is a better ordering from the following steps in CDA (the tokens @I12@, @I32@, …, etc. are new constants generated during the search process):

```
(1)  execute (?Person, name, 'George_VI /Windsor/') and
     get ?Person = @I32@;
(2)  get freq(@I32@, subject) = 8. Execute (@I32@, spouseIn,
     ?Family) and get ?Family = @F12@
(3)  get freq(@F12@, object) = 4. Execute (?Daughter, childIn,
     @F12@) and get ?Daughter = @I52@ or @I53@
(4)  get freq(@I52@, subject) = 7. Execute (@I52@, sex, F) and
     get True.
(5)  Execute (@I52@, name, ?Name) and
     get ?Name = Elizabeth_II Alexandra Mary/Windsor/.
(6)  get freq(@I53@, subject) = 7. Execute (@I53@, sex, F) and
     get True.
(7)  Execute (@I53@, name, ?Name) and
     get ?Name = Margaret Rose /Windsor/.
```

According to CDA, in each step we execute the clause containing constants of the lowest frequency found so far. In contrast, ISA never evaluates the remaining clauses the relative ranking of which with regard to given heuristics will change dynamically in the execution when the variable instantiation brings new constants into the clauses to be evaluated. The optimal search order for this query turned to be the original one (Query 0). Note that the frequency counts for some high frequency words are pre-collected while those for the low-frequency words (such as those in the italic fonts) can be acquired easily during the execution with little cost since there is no way to foresee their appearance before the execution. These counts are recorded for reuse.

The real execution time verified our analysis: to get all correct results, Query 0 took 10.21 seconds while Query 1 took 32,269.9 seconds. Since the deployment of CDA algorithm, we have never yet encountered any case that was terminated due to overtime.

Query broadening

OSSS offers query broadening by use of equivalence, generalization, specialization, and inverse relationships, as defined in OWL relationships. We chose not to offer full inference. Inference within the search would be problematic on several grounds. A standard for OWL rules did not exist yet. It would likely void any promise of a latency bound on queries. It would also make the results of the search very dependent on the ontologies encountered by the SSAs—it is possible that the rules within two different ontologies could lead to contradictory results. Use of standard OWL relationships is a good compromise for extending the search beyond the literal terms stated by the user.

Use of OWL relationships is extra functionality that costs additional time, so is placed under user control. Some examples of OSSS query broadening by use of equivalence, generalization, specialization, and inverse relationships are shown in Figure 4. In the basic and expert GUIs, the user selects “standard search” or one of the OWL relations to follow. The standard search is always done first. A relationship is followed afterwards, if one was selected. A pattern we observe is that a standard search is tried first and returns no results. The user realizes that the data uses a representation other than what he or she thought, and re-queries with one of the relations. If a query for phone number fails, for example, use of the specialization relation will turn up the area code, phone prefix, and phone suffix when that is how the tuples were encoded.

Example of an equivalence search:

```
Query to OSSS: (?PERSON work-address ?ADDRESS)
OSSS repository contains:
      (Bob work-at PARC) and
      (work-address samePropertyAs work-at)
Answer by OSSS:
      (Bob work-at PARC)
```

Example of a generalized search:

```
Query to OSSS: (Bob workPhoneExtensionNumber ?NUMBER)
OSSS repository contains:
      (Bob workPhoneNumber 650-999-1234-X256) and
      (workPhoneExtensionNumber subPropertyOf
            workPhoneNumber)
Answer by OSSS:
      (Bob workPhoneNumber 650-999-1234-X256)
```

Example of a specialized search:

```
Query to OSSS: (KSDivision produces ?PRODUCT)
OSSS repository contains:
      (InferenceEngineGroup produces InferenceEngines)
      and
      (InferenceEngineGroup subClassOf KSDivision)
Answer by OSSS:
      (InferenceEngineGroup produces InferenceEngines)
```

Example of an inversion search:

```
Query to OSSS: (?COMPANY employs Paul)
OSSS repository contains:
      (Paul work-for Stanford) and
      (work-for inverseOf employs)
Answer by OSSS:
      (Paul work-for Stanford)
```

Figure 4. Examples of query broadening

The presence of the samePropertyAs, subPropertyOf, subClassOf and inverseOf clauses are critical to the success of the searches. They enable the OSSS search engine to look beyond the original query. Ontology developers should be encouraged to use these OWL relations to bridge between the terms they defined and other ontologies. Sometimes, OSSS also computes the transitive closure of these relations and caches the results for runtime use. Future work should consider how far to do this for generalization and specialization where there may be deep taxonomies in the data. If one follows these relationships too far, the results start to have less relationship to the original query. The author of the ontologies should have the best knowledge about this. Query broadening demonstrates a major difference between semantic search and traditional search engines. Some of these engines now use very simple natural language morphological techniques such as stemming and alternative spellings to broaden a query, but the search is basically

dependent on the exact words used in the search. OSSS is able to broaden semantically so that the success of the search is not so much at the mercy of the exact terms used.

OSSS Implementation

OSSS is implemented in Java and SWI Prolog on SUSE Linux. (Interesting note: we had to switch from Red Hat to SUSE because Red Hat does not appear to use swap space correctly when there is 1 Gb of RAM and the program uses most of it.) The database can be any relational database as we only use the basic features. We use the maxDB version of mySQL. Our standard server is now a 1.1 GHz Pentium with 1 Gb of RAM and 1 Gb of swap space, but most of our performance studies were done on a 500 MHz machine.

The GUIs are all JSP applications that run on tomcat 4. We have very little reliance on client-side scripting. This allows the maximum range of users to access OSSS services.

The back-end component uses the HP Jena2 parser for OWL. The runtime component uses a scheduler for subscribed queries and mailer for sending new results to subscribers. Both are third-party, open source components.

There are both Java and Web Service APIs for query, string completion, publish and subscribe functions.

Using the semantic search service

In the long run, we regard the main user of semantic search to be intelligent agents. There being few of these as yet, we need to provide means for humans to pose semantic queries in the short term. However, this is a difficult task for non-logicians. The syntax is alien and much less forgiving than natural language, it is hard for humans to restrict themselves to binary predicates, and humans will not remember the specific terms defined in many specific namespaces.

We created three GUIs to explore different issues of user interface for semantic query. The basic interface helps the user formulate queries in simple (subject, predicate, object) format, ignoring the namespaces for each term. The search engine treats the omitted namespaces as wildcards. The expert interface lets the user specify every aspect of the query in detail, including the specification of namespaces for every term. Then a very experimental GUI explores use of highly restricted natural language for query which employs regular English words and a set of question words such as What, Who, When and Where .

All three GUIs are built as user agents that sit on top of the Java APIs.

A basic query interface

This interface addresses the fundamental issue of formulating a conjunctive query of triples. The target user is the technically inclined early adopter. Query construction proceeds by the following steps, as shown starting with Figure 5.

1. The user enters subject, predicate, and object for each tuple on separate lines. Items starting with “?” are taken as variables.
2. As shown in the figure, the user may enter the first few characters of what they think the term is, and OSSS will populate a pull-down menu of matching terms.

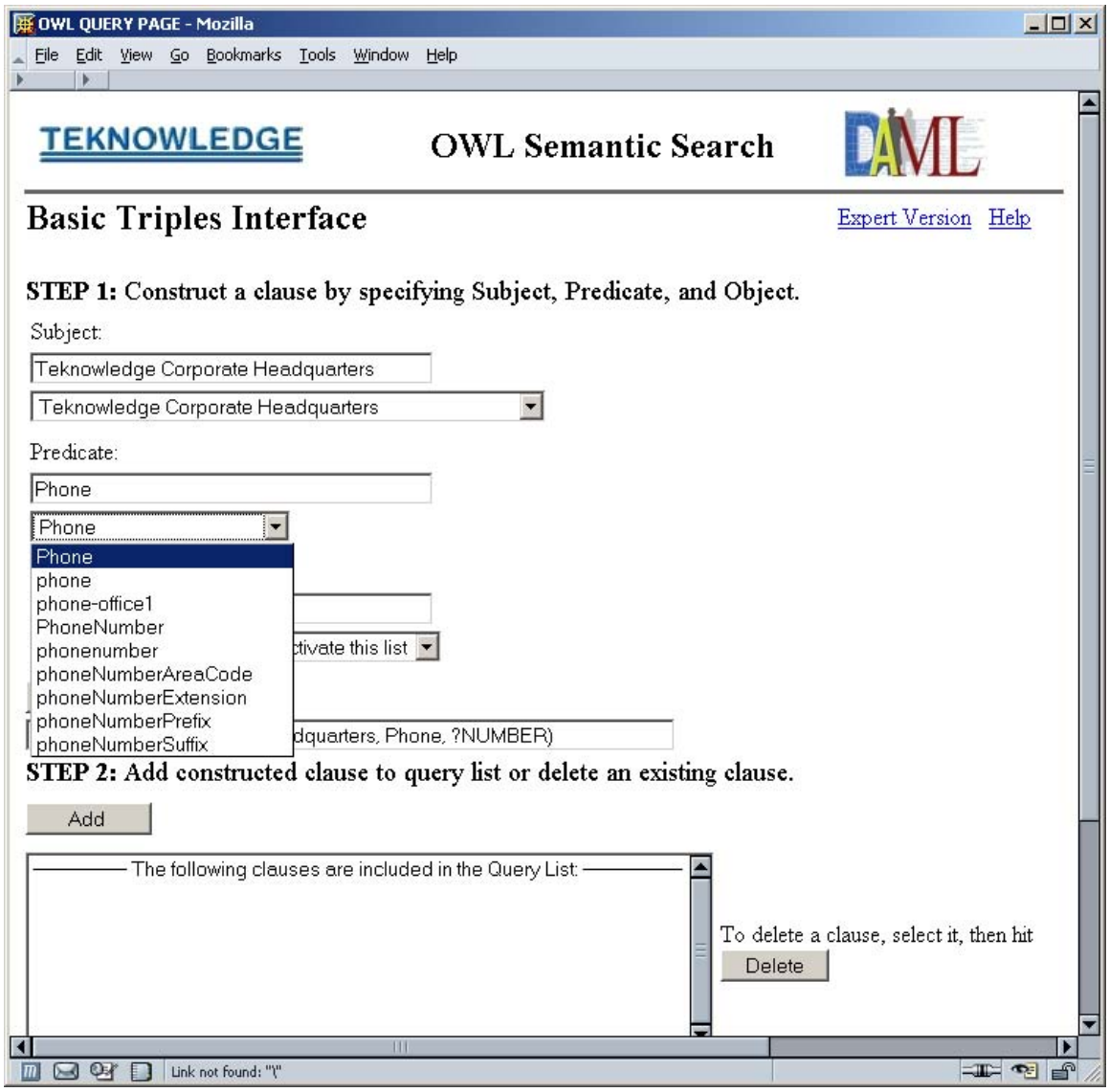

Figure 5. Basic interface, showing string completion

3. Going on to Figure 6, clicking the Construct button creates a full triple from the pieces entered by the user. If desired, the user can type this triple in directly, and not fill in the subject, predicate, and object slots.
4. Clicking the Add button adds this triple to the query. The user cannot type the full query in directly. He or she builds it up by adding and deleting triples.
5. Finally, the user selects the type of query (standard, or one of the types of query broadening explained above) and submits it.

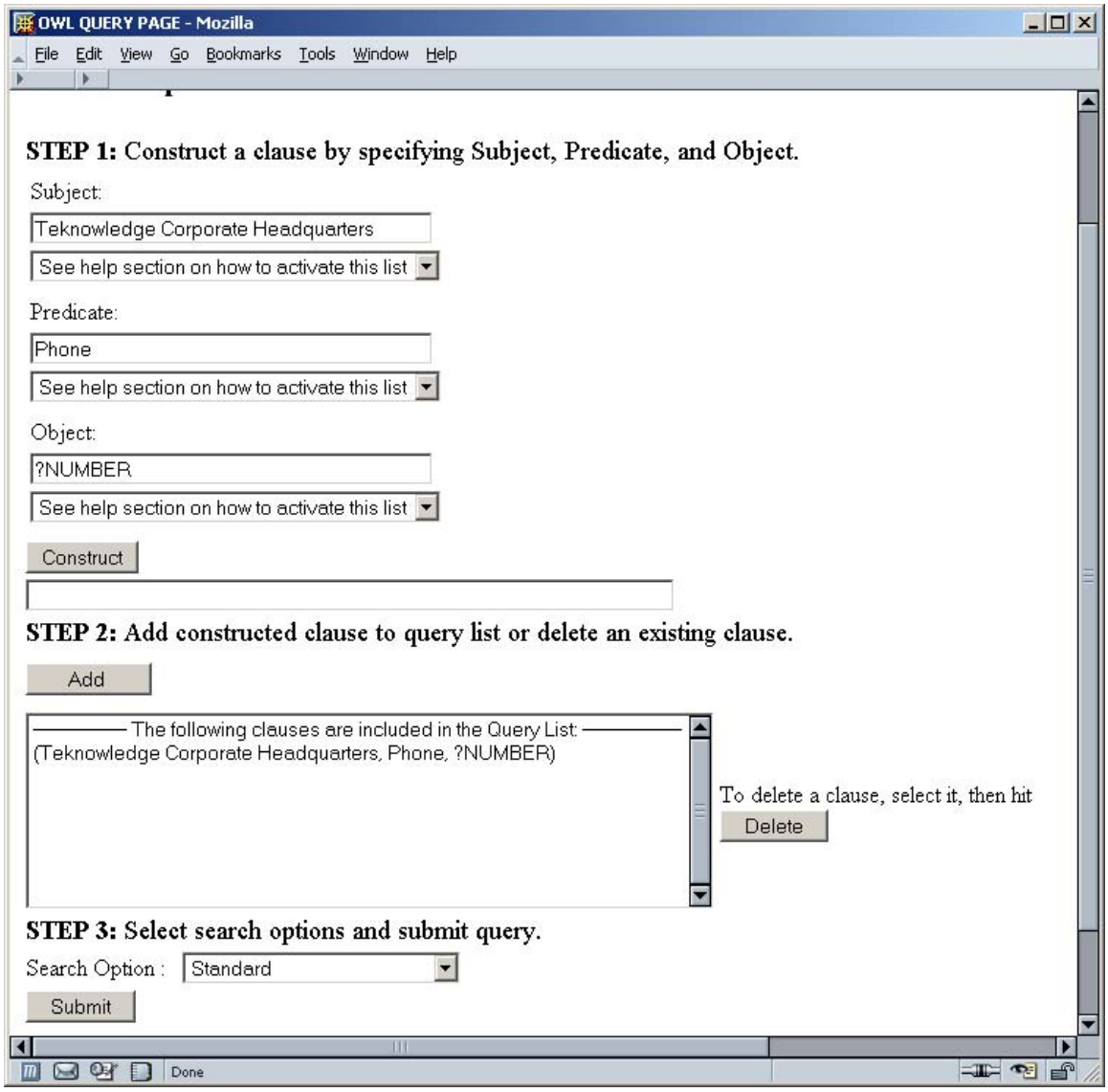

Figure 6. Basic interface showing simple query

Submitting the query results in a page of the first 10 results, as in Figure 7. If more results are wanted (and are available), the bottom of the page will have a Next Results button that retrieves the next batch of 10.

Figure 7. Search results

All results are links back to the source file and markup. The first URL for (Jena, name, Jena) leads to the following file:

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-
syntax-ns#" xmlns="http://www.daml.org/tools/tools-ont#">
  <Project rdf:ID="DAML">
    <name>DAML</name>
  </Project>
  <Tool xmlns="" rdf:ID="RDF_API">
    <name>RDF API</name>
    <description>Java RDF parser, from Stanford.  This is
      merging with SiRPAC.</description>
    <site>http://www-db.stanford.edu/~melnik/rdf/api.html
    </site>
    <interface>java_api</interface>
```

```
    <category>RDF Parser</category>
    <project rdf:resource="#DAML"/>
  </Tool>
```

An interface for expert users

This interface is designed for the "power user" who is well familiar with OWL and wishes to control all details of the query.

An expert query starts by identifying the default namespace(s) for the query. This is Step 1 in Figure 8. In addition to the string completion offered in the basic GUI, the user can ask to see a list of all namespaces known to the system from crawling, and when a namespace is selected, to get a list of all the terms it defines.

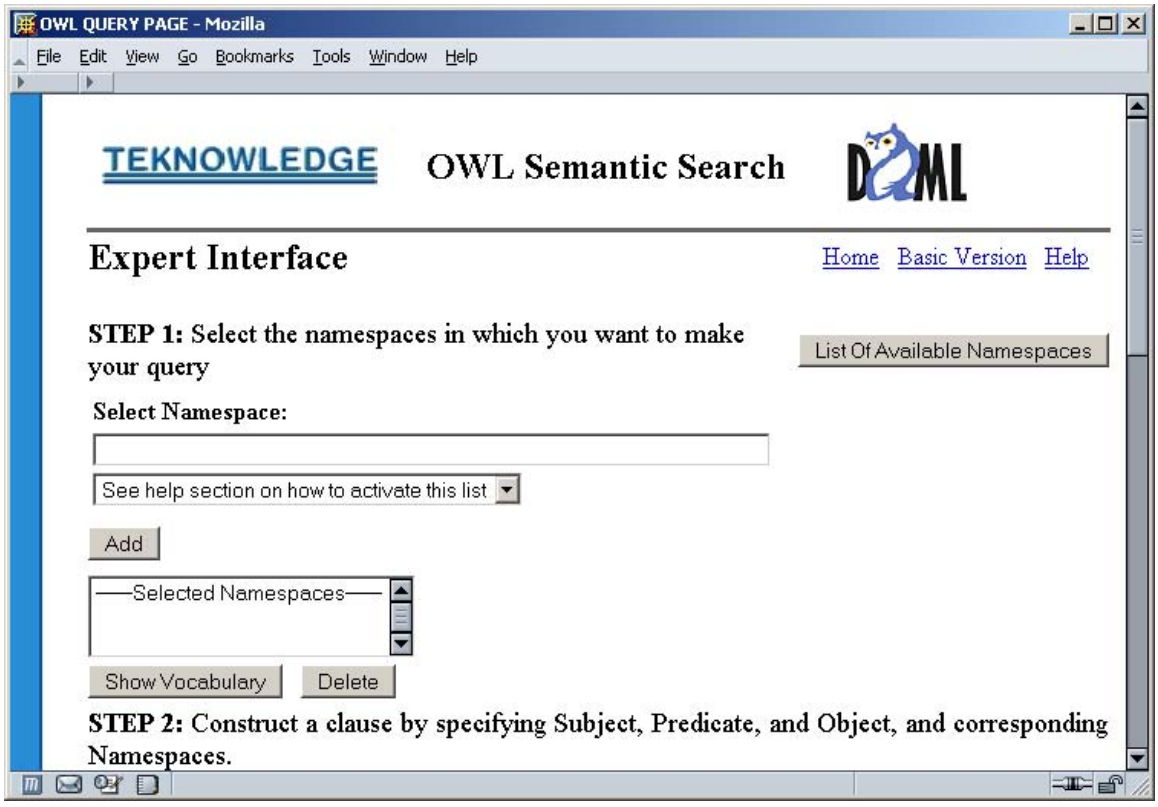

Figure 8. Namespace selection in the expert GUI

The next step is to create the next triple for the query (see Figure 9) but with more control than in the basic interface. The user may specify the namespace of each part of the triple separately. Clauses are otherwise defined, added to, or deleted from the query in the same way as the basic interface.

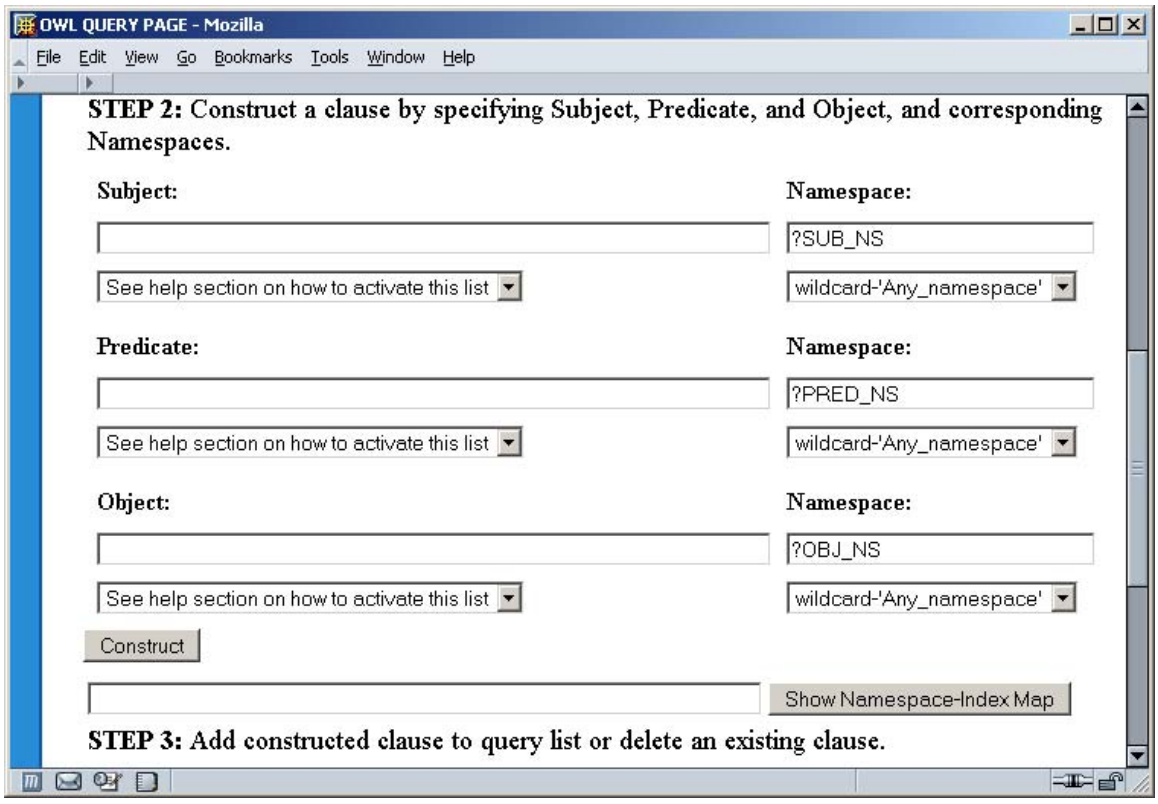

Figure 9. Constructing clauses

Once the query has been constructed, Figure 10 shows submission details. The user can request the type of search as before, but can also set query parameters. Instead of getting batches of 10 results, the user can specify each and every batch. Finally, a query timeout can be defined. 300 seconds is a very generous timeout considering our test queries take 10 seconds or less.

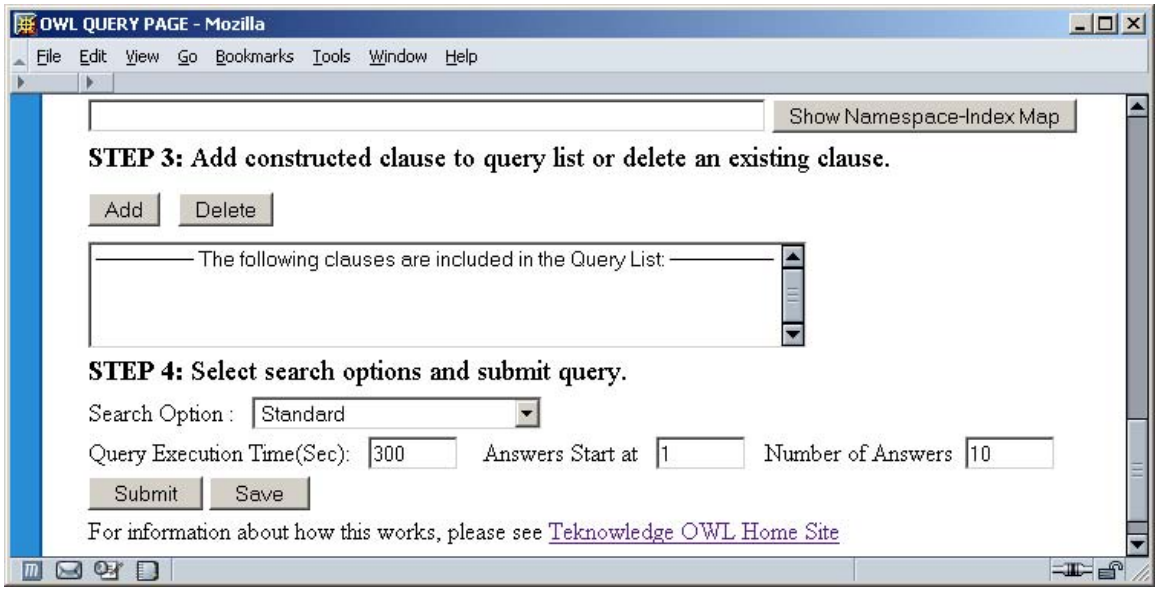

Figure 10. Submitting the expert query

The results page for the expert interface is the same as for the basic interface.

An experiment with a natural language interface

This interface is an experiment to address the less-technical user. It uses a journalistic Who, Where, What, Why as query variables, and a form of restricted English. The query shown in Figure 11 can be paraphrased as "Who wrote (something with the title of) Temporal Logic Programming?"

The user still has to decompose his or her question into triples, but some of the formal syntax has been removed. The pull-down menus offer Who, What, Where, and What as standard variables, since those are often what variables in simple queries are about. Two other features make this more like natural language and less like OWL. The first is that this GUI also demonstrates a hub & spoke approach to ontology. We mapped over a dozen other ontologies to SUMO plus MILO as a hub ontology. The equivalence statements that the mapping adds means all query terms assumes the SUMO namespace. The user does not have to worry about that issue. The second feature is use of WordNet to SUMO mappings. If the user enters some regular English words, each word is first converted into its normal form through a sequence of morphological processing and then the whole clause is mapped to a SUMO logic relation via WordNet to SUMO mappings. In this way, the user does not have to know the terms in the namespace and their exact spelling.

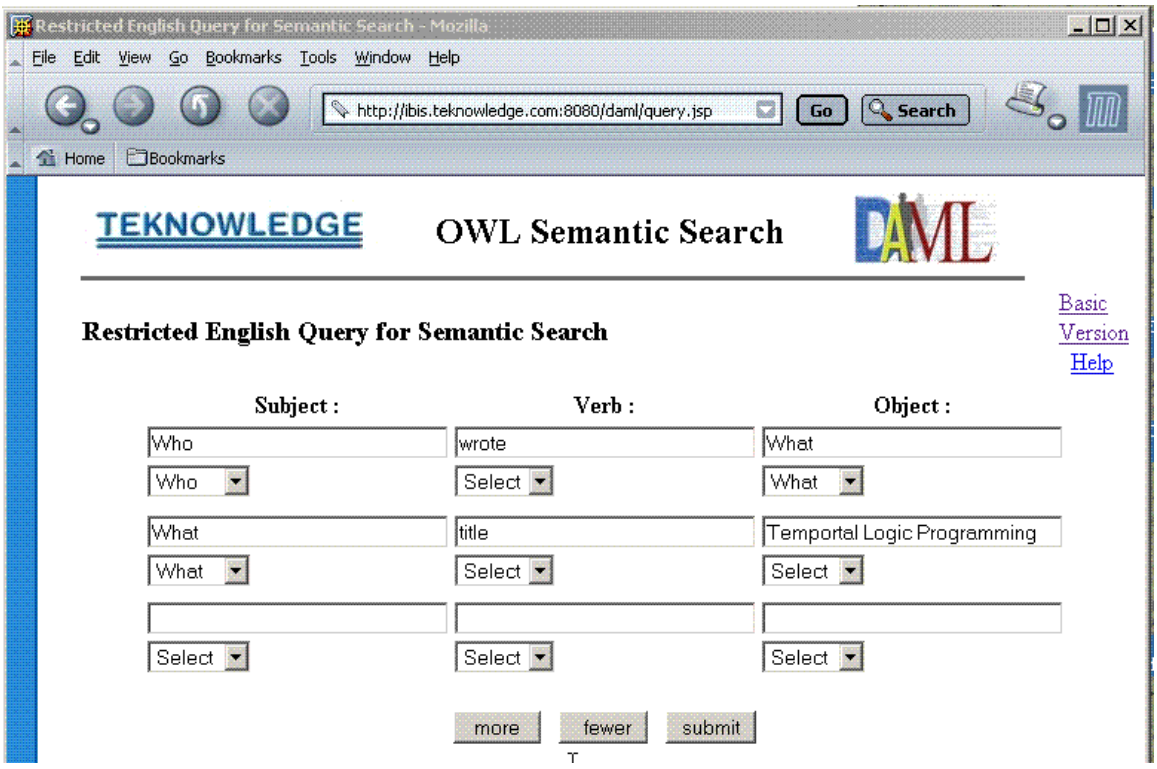

Figure 11. A query using the controlled English interface

It is instructive to compare this OSSS GUI with a more complete natural language interface. Many of the ideas for this GUI come from our CELT project (Controlled

English Logic Translation). CELT [Murray04] accepts queries posed in a controlled but fairly expressive subset of English called ACE. It uses the WordNet mappings from English words to concepts in the SUMO ontology. The English query is converted into a logical form, which is then answered by query to a theorem prover.

This restricted English GUI does not allow entry of queries in free text but does use the WordNet techniques and makes query to a logical engine. CELT featured deep semantics, good linguistic processing of the free text, but a small and static vocabulary (like most of the NLP systems). By contrast, this OSSS GUI does limited linguistic processing but is able to utilize a far larger vocabulary, is dynamic, and in some sense unbounded. However, note that the current Ibis system currently has only terrorism and bibliography content.

Figure 12 show the results of a query "Who wrote Temporal Logic programming. The original query, (Figure 11) used "wrote" and "title" as predicates. Since the ontologies for the bibliographic data in this repository have been mapped to SUMO and MILO, the source data does not have to be expressed using those terms.

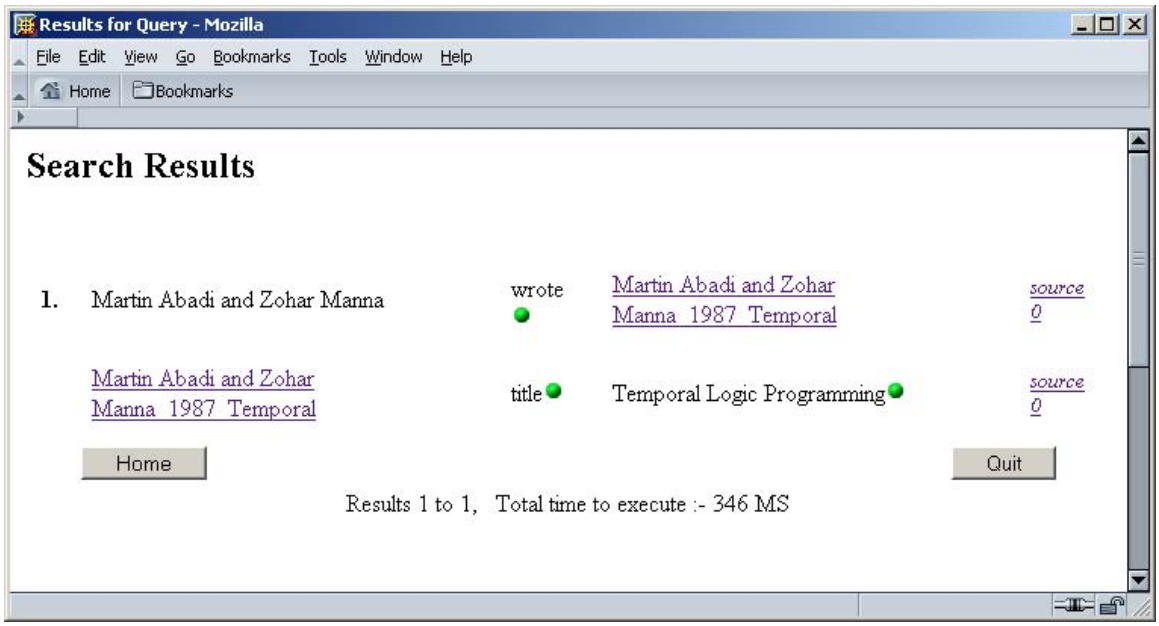

Figure 12. Query in original terms

This user interface allows the user to access the data's original terms. The small green balls are a toggle between the hub ontology and the original spoke ontology of the data. Figure 13 shows the result of toggling both predicates. For example, the query "Who wrote What" is answered by data expressed as "Who authors What"

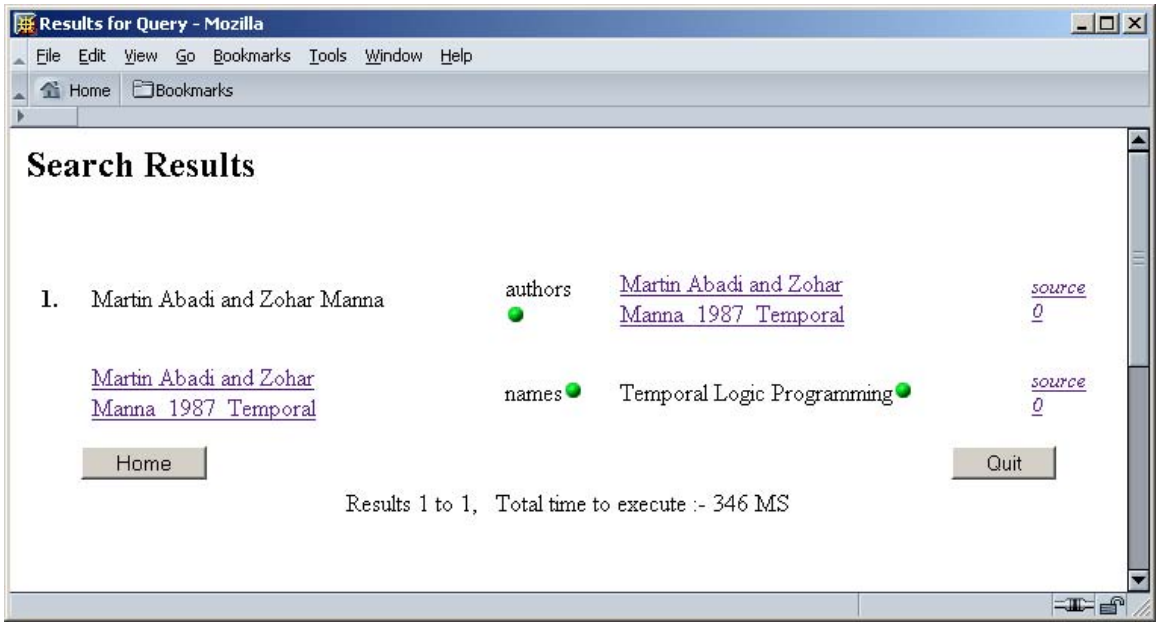

Figure 13. Query in terms used by source data

This interface also experimented with compiling some data into a cache as an optimization. As part of a batch crawl, it precomputes and caches generalizations and specializations to avoid doing this during a query. This is a classic space vs. time optimization. We currently do the full transitive closure. We may in the future experiment with limiting this to avoid useless transitions.

Publish, subscribe, and notification functionality

We implemented a publish/subscribe facility to extend the utility of OSSS. The publish feature allows the user to enter the URL where the crawler may find some new OWL markup. One can also enter a previously crawled URL if a repository update is desired. In either case, OSSS crawls the URL immediately (or as soon as possible) and indexes the triples found there. The runtime system is updated transactionally to make the new data available.

The subscription facility addresses the other end of the information workflow. When a user logs into the subscription system, he or she creates a query using the standard query interface. However, a Save button is offered along with Submit. The Save button takes the user to an edit query screen as shown in Figure 14 which collects information for completing the subscription. In this screen, the query is shown in translated, internal form. If the user wants the query to be tested upon submission of a URL, it collects the specific URL. Finally, in addition to the usual query specifications, the user enters the frequency for checking the query. There are both schedule-based specifications (from two minutes for demo purposes up to Monthly) and event-based ones. Contents_Update checks subscribed queries whenever any data in the repository changed. URL_submission does the check whenever a document at a user-specified URL is updated in OSSS.

When a subscribed query is triggered, OSSS does a check on the results. If it differs from the old response in terms of the number of answers or in the word count of the response, it represents new information. A standard results page is generated and emailed to the saved address.

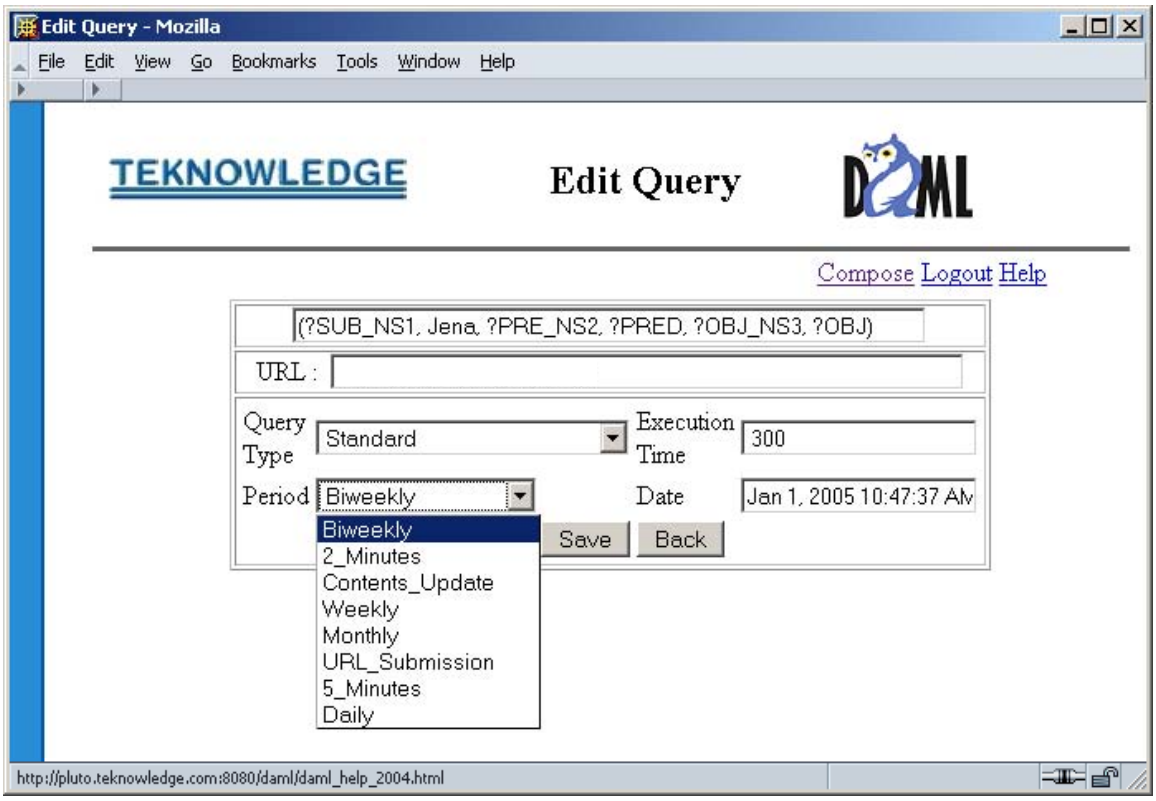

Figure 14. Editing a subscribed query

The publish and subscribe features further adapt OSSS to a dynamic environment. When user can publish new data to the system, there is less reliance on what a periodic crawl discovers. Subscriptions mean users are proactively notified of relevant information, on a schedule of their choosing. The combination promotes an active "infosphere" linking producers and consumers of semantic data.

Semantic translation

A central, but unsolved, problem for the Semantic Web is ontology mapping. Seen through the lens of search, the quality and quantity of answers to queries are restricted by the amount of data it can touch. The proliferation of namespaces limits search to small islands unless a general solution is found that can bridge between these islands of meaning.

One partial solution is the use of common upper ontologies. An upper ontology defines a set of ontological commitments and a framework of abstract concepts. It serves as an organizing principle when less abstract ontologies are derived from it. A good example is SUMO (Suggested Upper Merged Ontology) developed at Teknowledge and now a candidate in the IEEE standards process. SUMO defines a fairly standard first-order logic representation, and makes its commitments in how time is handled, metrics

and measures, set theory, etc. It defines Entity as a top-level concept, specializes that to Physical and Abstract, and goes from there. By design, SUMO is held to about 1,000 concepts, not counting the rules that give them further meaning.

We derived MILO (the MidLevel Ontology) from SUMO. MILO was constructed by examination of the well-known 1,000,000 word Brown Corpus of written English as mapped to WordNet [Niles04]. Every WordNet synset (concept) that appears at least three times was made into a MILO concept, if it did not already appear in SUMO. In this way, we ensured that MILO was grounded in real-world experience. MILO adds a little over 2,000 additional concepts. The combination of SUMO and MILO is an excellent base because it is small enough to remain understandable and maintainable , it is general enough to cover all domains, and contains a fringe of grounded concepts that ensures relevance in real-world problems.

We do not advocate that all OWL content be expressed in a SUMO-based ontology. To belabor the metaphor, upper ontologies such as SUMO help extend islands of semantic interoperability into continents. If two ontologies share a common upper ontology, there is probably sufficient commonality so that there is a good chance they can be mapped and mutually interoperate. If they don't share such a common understanding, it is very likely a human will have to do the large-scale conceptual matching before an automated process can start to assist. It is also possible the two ontologies will differ so much in approach that it may be prohibitive to try to map between them.

The "natural language" GUI discussed above is a demonstration of the upper-ontology approach. SUMO is used as the hub of a hub & spoke approach. We created N mappings between SUMO and other ontologies instead of N^2 mappings between all pairs of ontologies. Each mapping is represented as sets of OWL equivalence relations. All queries can then be made in SUMO's vocabulary and results reported in the same.

The lexical ontology mapper

When it became clear that creation of an automated ontology mapper is long-term research that will last longer than the DAML program, we sought a short-term stopgap. We created a semi-automated mapping utility [Li04] based on lexical methods, following many of the same heuristics that humans use when mapping between ontologies.

The lexical ontology mapper (LOM) compares the terms in two OWL ontology files. Its output is a list of suggested correspondences and their estimated similarities. A number of heuristics are tried, in order, until a sufficient match is found. The current algorithm is:

(1) Whole word matching. A candidate pair is converted to lower case and matched by exact string comparison.

(2) Word constituent comparison. Many terms have multiword names. After canonicalization and parsing into constituents based on case, use of stop words, and separator characters, the terms undergo a simple morphological analysis. Each constituent in one term is then string matched to constituents in the other. This procedure discovers correspondences such as "written-by" and "wrote".

(3) WordNet synset matching. This is similar to the last match, but maps each constituent into its corresponding synset before comparison to interject some semantics into the match. The pair with the largest number of overlaps in synsets is selected. Synset matching discovers correspondences such as "AutoCare" and "car-maintenance".

(4) Ontological concept matching. If a candidate pair has not already been matched, each constituent is mapped to its WordNet synset, and from there to its corresponding ontological concept in SUMO plus MILO. This level of analysis can discover correspondences such as between "Armored_personnel_carrier" and "Tank" since both are types of military vehicle.

LOM is extensible. It is relatively easy to integrate new matching heuristics into the existing framework.

LOM is intended as a utility that helps the human in mapping. It suggests plausible matches but the human still has to vett them and fill in matches LOM was unable to make. For comparing a file of *n* terms with one of *m* terms, LOM's runtime is O(*nm*), however each step is simple and requires little or no inference. Because of this, LOM is more efficient with large ontologies than reasoning-based approaches.

Semantic translation agents

We created agents that provided hand-built mappings between specific ontologies. We demonstrated a hub & spoke approach for dealing with multiple namespaces, and created the LOM utility. One can also think of search broadening along equivalence relations a form of translation agent.

We also provide advice to the content creator. It is critical for creators to take some responsibility for making their content interoperable and understandable beyond a (possibly idiosyncratic) namespace. Seeding the content with equivalences to other namespaces can be of enormous help. Adoption of a layered approach such as SUMO plus MILO, plus domain ontologies is a complimentary means of widening the applicability of the content.

For all these approaches, it is important to remember that exact answers to queries can often only be guaranteed within a single ontology. Matching with equivalence and inverse relations can cross ontologies exactly, but in most cases that luxury is not available. Matching one concept to a more general concept results in a less exact match. And when mapping between two dissimilar ontologies, there is great opportunity to lose nuances of meaning.

Other contributions

Professor Hendler called for DAML teams to help seed the Semantic Web with content. We generated over 12,000 dense, marked-up pages from WordNet. But more substantially, we made a considerable ontological asset available in OWL format. Throughout the program (and in a variety of markup languages ending with OWL), we provided translations of the SUMO upper ontology, our MILO midlevel ontology and 13 of our substantive domain-level ontologies that depend on SUMO and MILO. These are ontologies originally developed for military and intelligence customers on other projects.

Our ontologies are developed in full first-order logic and stored in KIF. Ontologists use our Sigma Workbench, which integrates the Vampire theorem prover from the Univ. of Manchester. Ontologists use Sigma to load ontologies, browse them, check consistency and redundancy, and query them. Sigma is available free to Government users.

All these ontologies were posted on DAML.org, and updated OWL versions posted to SemWebCentral.org. Note that since SWRL had not been adopted yet, the OWL translator drops the axioms found in the original KIF.

Analysis

ASCS benefits

- Access to semantically marked-up data is a key enabler of the Semantic Web or the larger semantic infosphere of Web pages and other data sources.
- Semantic queries return the answers that were being looked for, not a further "homework problem" to find the real answers.
- Query broadening reduces dependence on knowing the exact terms to use in the query, or knowing precisely what you are looking for.
- Semantic queries pull together answers even when the source data resides in many very different data sources. The query does not have to know what these sources are ahead of time.
- The publish and subscribe features help ensure the data is up-to-date, and that interested parties are notified of relevant change in a timely way.
- The LOM reduces the work required to create an ontology mapping, and may suggest correspondences a human might overlook.

Key innovations from the ASCS project:

- A semantic search engine that works on OWL markup rather than keyword searches. OSSS strikes a good balance between fast performance, query expressiveness, and use of inexpensive hardware.
- Optimization by continuous query analysis that results in over 1000x speed up for many complex queries.
- Query broadening by use of OWL relationships shows that a very little bit of inference can go a long way for the user.
- A practical and fast method of discovering and rating correspondences between terms in OWL-based ontologies
- A distributed-agent architecture for search that addresses the presence of numerous namespaces

Scalability

As of Fall 2004, OSSS indexed about 7.75 million tuples on one machine. This required 500 Mb of RAM for the repository and 400 Mb for the string completion trees. In this version (the standard and expert interfaces), there is no caching of OWL relationships.

Just looking at tuple capacity, there is no need for the string completion trees to be served from the same machine as the search repository. A server with 2 Gb of RAM is not an unusual machine. Assume 256 Mb goes to the operating system and the program. That leaves 1.75 Gb of RAM for the search image and data. This is 3.5x the size of the

present system, or 27 million tuples. The DB and operating system can handle this, can the Prolog?

Assume there are 25 million seven-tuples in the repository to search. That is not a situation for which standard Prolog knowledge-base hashing will shine. If you index on the function only, everything falls into one hash bucket. SWI allows the user to modify the hash function to use specified arguments. We use this to index all the arguments. The hash function may be slightly more expensive to compute but brings KB access back to better-than-linear performance. We were not able to do experiments to find the practical upper limit to the size of a SWI image before DAML funding was cut, but believe 2 Gb is a good target. (If the Prolog will support images many times larger than this with reasonable performance, two and four CPU machines with large RAM are still moderately priced.) The point is that two commodity class PCs will not support search of the entire Semantic Web or enterprises with Tb sized databases which can be exported as OWL, but should be sufficient to power semantic search for one enterprise and its data.

A second analysis is based on the fact that we search a good part of the Semantic Web and have done periodic crawls for over two years. (We have not devoted the resources to crawling to guarantee that we have found every page with some kind of markup on it.) The 7.75 million tuples mentioned above represents about 90,000 pages. More accurately, there are about 20 "database pages" (such as all SIC codes) that account for 2.3 million triples. These are characterized by simple ontologies but large numbers of instances. There are about 80,000 pages generated from WordNet that are static. We have found only about 4600 "real" Semantic Web pages. We did not include sites like Friendster or RSS blogs that minimally use RDF.

Discounting the few database pages, there are about 90 tuples/page on the Semantic Web. As per above, we estimate a single OSSS server should be able to handle 27 million tuples. That represents a growth of 19 million tuples, or at least 21,000 new pages. If the Semantic Web grows at 100% annually, one server should be able to provide search for the next five years. The architecture supporting this consists of a single processor PC with 2 Gb of RAM, a similar one for the database and the string index, and a third dedicated to crawling. At 2004 prices, this is still on the order of $5,000 in total.

The string completion index is a tree structure that discriminates ten-character strings, starting with the first character. We observe this grows slower than the number of new namespaces. People tend to reuse a small number of names for objects. Even so, this structure need not be the scalability bottleneck. It can be split over several machines, A-K on one, L-R on another, and so forth. The main bottleneck will be the tuple repository as long as it remains in memory.

Desirable near-term systems work

This section discusses work an ASCS successor project will likely consider. These points are not research but systems work that will adapt ASCS to operations in some specific application environment.

Adapt or build a hardened crawler that is able to deal with the myriads of problems found on the Web. These range from access problems, to poor markup, to complex rings of pointers.

We assume that specific data feeds from sensors, databases, or other controlled sources will have fewer problems. However, in the Semantic Web, the OWL content is available on the page. There may be a need for a crawler visiting a registered controlled data source to be able to ask for an OWL export.

The whole issue of security and access needs to be analyzed for any application. There should be a protocol for sites to grant access rights to crawlers, and perhaps including access to only parts of the content. There should be authentication and access control for both publishers and subscribers. Finally, the ASCS system should be hardened against crackers using it for exploits.

Further instrumentation and systems testing to characterize the performance envelope for a given set of features and parameter choices is highly desirable. Ideally, this instrumentation will be a package of utilities for system administrators to use in tuning ASCS implementations in their specific environment.

Now that OWL and related standards are becoming more mature, ASCS should adopt OWL-S and current implementations of matchmaking to enable SSAs to find each other.

Finally, the server currently runs in Linux. It would be useful to also have a Windows version.

Areas for further research

One immediate need is to assist humans in formulating queries. We found many ways to help users formulate OWL queries but did not solve the problem that such queries are quite unnatural to most users. An ideal solution might be one that lets the user state the query in a restricted natural language and then suggests reformulations in terms of triples and the user's favorite namespaces.

An important question is the amount of expressiveness which can be allowed in the query language while retaining small and predictable response latency. Now that rules are becoming part of the OWL language, queries might utilize the rules found in the ontologies used to express specific data. Inference over these rules will allow even more intelligent answers to be returned, but at the risk of seemingly simple queries timing out due to the inference triggered by the query. Worse yet is when higher expressiveness results in "falling off" the computational cliff for no reason discernable by the user. Use of inference in the various ontologies encountered during the search can lead to contradictory results where the ontologies are not consistent. Clearly, a good number of tradeoffs and experimentation is required before this kind of search is common.

The issue of scaling up in repository size and query complexity will become increasingly pressing with the growth of marked-up data. In a distributed-agent architecture, more work is needed on the combination of query results. The assumption is that each SSA does its search in isolation and we need to merge the results. This will be subject to a quality vs. time tradeoff. Can it be structured as an anytime algorithm able to return increasingly better answers as it is given more resources? More time means that more SSAs might be enlisted to contribute. It remains to be verified if this necessarily results in an improved set of answers. This tradeoff should be under user control. Only they know if speed or completeness is more important for the current query.

We hypothesize that the content of individual SSAs will tend to have a specific focus. It will consist largely of what is expressed in one or a few ontologies, and/or it will have a domain focus. It could be possible that two SSAs advertise ability to search some topic, but their separate ontologies will lead to potentially contradictory results. Assume SSA-1 advertises geography expertise and is based on a flat-earth ontology while SSA-2 also advertises geography but is based on spherical geometry. If a different answer comes from each SSA, the two results may be contradictory but each is independent and true to its ontology. It can be a problem if the query was partitioned and partial answers come from mis-matching ontologies. Research is needed into economical ways for the merge process to detect this problem and make a decision on what to do about it.

It will be problematic to partition parts of a single repository to many servers and process in parallel. In this scenario, the different nodes do not simply supply additional answers to merge. All the nodes must be enlisted in a single computation. An obvious problem is the coordination of backtracking and continued search across many machines.

One possible way to break out of RAM limitations is a tight integration of search engine with a database. For ordinary Prologs, it is too expensive to query for data in the middle of a search, especially when there is extensive backtracking. An interesting avenue to explore would be to use a Datalog implementation, if a high performing, modern implementation exists, or other low-cost means of integrating database access. (One strong factor in PARKA's strong performance was a willingness to build the data access directly into the reasoning engine.) If a query could do a little analysis and swap in the triples it is likely to need, the result would be highly conducive to massive horizontal scalability. No machine would depend on loading the entire repository into RAM and the next query could be shunted to the next available server.

There is a problem along a separate dimension. The tuples gathered on one date are expressed in specific ontologies. However, ontologies evolve over time. Work is needed to detect ontology change and to propagate its effects into the repository. A set of instances may be restated if the underlying ontology changes. Recrawling that site and reindexing its tuples may be a good way to deal with that case. Any cached information based on inference from an ontology may have to be invalidated and recalculated. The more problematic case is when there is semantic creep. New instances are created based on a slightly different meaning for terms than earlier instances. This is a typical version control problem of the ontology development.

Conclusion

The ASCS project created a key building block of the Semantic Web, a semantic search engine that provides access to OWL data. The project investigated architectural issues from the global (an open-ended distributed agent system) to the local (the internal structure of individual nodes). Our work identified user interface and ontology mapping as important issues that might reduce the utility of the search engine, so we created solutions for both. The project legacy is the search engine, its interfaces, the ontology mapper, and its contribution to SemWebCentral as running code for early adopters to use.

References

[Li01]
Li, J., Pease, A., Barbee, C. 2001. "Performance of Semantic Search." Teknowledge Technical report, June 15, 2001.

[Li02]
Li, J., Pease, A. 2002. "Building the DAML Semantic Search Services." Teknowledge Technical report, June, 2002.

[Li04]
Li, J. 2004. "LOM: A Lexicon-based Ontology Mapping Tool" In *Proceeding of the Performance Metrics for Intelligent Systems (PerMIS '04)*. Information Interpretation and Integration Conference (I^3CON), Gaithersburg, MD.

[Murray04]
Murray, William R., Adam Pease, and Michelle Sams. 2004. "Applying Formal Methods and Representations in a Natural Language Tutor to Teach Tactical Reasoning." In *Proceedings of the Eleventh International Conference on Artificial Intelligence in Education*. Sydney, Australia: International Artificial Intelligence in Education Society.

[Niles01]
Niles, Ian, and Adam Pease. 2001. "Towards a Standard Upper Ontology." In *Proceedings of the Second International Conference on Formal Ontology in Information Systems*, 2–9.

[Niles04]
Niles, Ian and Allan Terry. 2004. "The MILO: A general-purpose, mid-level ontology." In *2004 International Conference on Information and Knowledge Engineering (IKE'04)*. Las Vegas, NV.